Variables in Memory

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Objectives

This lecture covers some of the different places variables can be kept in memory. By the end of lecture, you should be familiar with

- The function call stack
  - static and dynamic links, and how nested scopes are resolved
  - frame pointer, stack pointer
- how local variables are stored
- how object member variables are stored
- the difference between boxed and unboxed data

Global Variables

Global variables are kept on the system stack.
Consider `let i = 5 * 2`
Order of operations:

- First, `5 * 2` is evaluated.
- Second, `10` is place at the top of stack.
- Third, `SP` is decremented by one word.

Stack

Addr	*Value*	
8000	10	i
7996		SP
7992		
7988		
7984		
...	...	

What happens when
`let (j,k) = 10 + i, i + 2` is executed?

Answer

Stack

Addr	*Value*	
8000	10	i
7996	20	j
7992	12	k
7988		SP
7984		
...	...	

What if we now execute `let i = i + 4`?

Stack

Addr	*Value*	
8000	10	i
7996	20	j
7992	12	k
7988	14	i
7984		SP
. . .	. . .	

The first `i` is unreachable, so it's garbage.

There are plenty of languages that use this kind of access!

- Forth
- Postscript
- Assembly

These languages are called *stack-based*. They are suitable for small, cheap processors, low amounts of memory....

C-like languages have functions with these properties:

- Two layers of scope (local and global).
- Multiple, possibly variable number of parameters.
- Functions not first-class.

In order to call a function in a C like language, we need space on the stack for several things.

- Return address
- Arguments
- Pointer to previous stack frame
- Local variables

arg n . . . arg 1 dynamic link	frame pointer
local var 1 . . .	current frame
return address other stuff	
arg m . . . dynamic link	

- The return address points to the machine code of the calling function.
- The dynamic link points to the stack frame of the calling function.
  - Don't confuse them!!
- Registers, temporary values, etc. get put in the "other stuff" section.

Example

Stack before `foo(42)` is called.

```
1  int g = 10;
2
3  int inc(int i) {
4    int t = i + 1;
5    return t;
6  }
7
8  int foo(int q) {
9    int k = 12;
10   k = inc(k+q);
11 }
```

Addr	*Value*	
8000	10	g FP
7996		SP
7992		
7988		
7984		
...	...	

Stack before `inc` is called.

```
1  int g = 10;
2
3  int inc(int i) {
4    int t = i + 1;
5    return t;
6  }
7
8  int foo(int q) {
9    int k = 12;
10   k = inc(k+q);
11 }
```

Addr	*Value*	
8000	10	g
7996	42	q
7992	8000	FP dynamic Link
7988	12	k
7984	line ??	foo's caller
7980		SP
...	...	

Calling `inc`.

```
1  int g = 10;
2
3  int inc(int i) {
4    int t = i + 1;
5    return t;
6  }
7
8  int foo() {
9    int k = 12;
10   k = inc(k+q);
11 }
```

Addr	*Value*	
8000	10	g
7996	42	q
7992	8000	dynamic Link
7988	12	k
7984	line ??	foo's caller
7980	54	i
7976	7992	FP dynamic Link
7972	0	t
7968	line 10	return
7964		SP
...	...	

Stack before `inc` returns.

```
1  int g = 10;
2
3  int inc(int i) {
4    int t = i + 1;
5    return t;
6  }
7
8  int foo(int q) {
9    int k = 12;
10   k = inc(k+q);
11 }
```

Addr	*Value*	
8000	10	g
7996	42	q
7992	8000	dynamic Link
7988	12	k
7984	line ??	foo's caller
7980	54	i
7976	7992	FP dynamic Link
7972	55	t
7968	line 10	return
7964		SP
...	...	

Stack after `inc` returns.

```
1  int g = 10;
2
3  int inc(int i) {
4    int t = i + 1;
5    return t;
6  }
7
8  int foo(int q) {
9    int k = 12;
10   k = inc(k+q);
11 }
```

Addr	*Value*	
8000	10	g
7996	42	q
7992	8000	FP dynamic Link
7988	55	k
7984	line ??	foo's caller
7980		SP
...	...	

- Many languages have *nested scope*... functions can be defined within functions.
- Pascal is the most famous.
- OCaml (and most functional languages) let you do this too.

```
1  let rec foo a =
2    let t = 10 + a in
3    let bar b =
4      let u = 20 in a + u + b + t + foo 9
5    let baz c =
6      let v = 30 in a + v + c + t + bar 5
7    in baz 4
```

- We now need a pointer to the *parent scope's* stack frame.

arg n ... dynamic link	frame pointer
local var 1 ...	current frame
return address *static link* other stuff	
arg m ... dynamic link	

- The static link points to the stack frame of the parent function.
- The dynamic link points to the stack frame of the calling function.
  - Don't confuse them!!


```
1  let rec foo a =
2    let t = 10 + a in
3    let bar b =
4      let u = 20 in
5         a + u + b +
6          t + foo 9
7    let baz c =
8      let v = 30 in
9         a + v + c +
10         t + bar 5
11   in baz 4
```

Addr	*Value*	
8000	7	a
7996	9000	FP Dynamic Link
7992	17	t
7988	Line ?	Return
7984	9000	Static Link
...	...	

Call `foo 7`. SP always points to the first empty spot.

Example


```
1  let rec foo a =
2    let t = 10 + a in
3    let bar b =
4      let u = 20 in
5         a + u + b +
6          t + foo 9
7    let baz c =
8      let v = 30 in
9         a + v + c +
10         t + bar 5
11   in baz 4
```

Your turn! What happens when `bar` is called?

Addr	*Value*	
8000	7	a
7996	9000	Dynamic Link
7992	17	t
7988	Line ?	Return
7984	9000	Static Link
7980	4	c
7976	7996	FP Dynamic Link
7972	30	v
7968	Line 11	Return
7964	7996	Static Link
...	...	

Addr	*Value*	
8000	7	a
7996	9000	Dynamic Link
7992	17	t
7988	Line ?	Return
7984	9000	Static Link
7980	4	c
7976	7996	Dynamic Link
7972	30	v
7968	Line 11	Return
7964	7996	Static Link
...	...	

Addr	*Value*	
7960	5	b
7956	7976	FP Dynamic Link
7952	20	u
7948		
7944		
...	...	

Addr	*Value*	
8000	7	a
7996	9000	Dynamic Link
7992	17	t
7988	Line ?	Return
7984	9000	Static Link
7980	4	c
7976	7996	Dynamic Link
7972	30	v
7968	Line 11	Return
7964	7996	Static Link
...	...	

Addr	*Value*	
7960	5	b
7956	7976	Dynamic Link
7952	20	u
7948	9	a
7944	7956	FP Dynamic Link
7940	19	t
7936	Line 6	Return
7932	9000	Static Link
...	...	

Objects

- The stack is used for local variables, parameters, and scalar data.
  - Integers, references, etc.
- Objects, and other “large” (non-scalar) data is usually placed on the heap.
  - Java: objects
  - OCaml: user-defined types, objects
  - C: things allocated with `malloc()`
- Heap allocated data persists across function calls.

Heap Example

```
1 let xx = 10 :: 20 :: []
```

Stack

Addr	*Value*	
8000	10000	xx
7996		
7992		
7988		
7984		
...	...	

Heap

Addr	*Value*	
10000	10	Head of First Node
10004	10008	Tail of First Node
10008	20	Head of Second Node
10012	0	Tail of Second Node
10016		
...	...	

Heap example 2

```
1 let xx = 10 :: 20 :: []
2 let x::xs = ...
```

Stack

Addr	*Value*	
8000	10000	xx
7996	10	x
7992	10008	xs
7988		
7984		
...	...	

Heap

Addr	*Value*	
10000	10	Head of First Node
10004	10008	Tail of First Node
10008	20	Head of Second Node
10012	0	Tail of Second Node
10016		
...	...	

Heap example 3

```
1 let xx = 10 :: 20 :: []
2 match xx with x::xs = ...
```

Stack

Addr	*Value*	
8000	10000	xx
7996	10	x
7992	10008	xs
7988		
7984		
...	...	

Heap

Addr	*Value*	
10000	10	Head of First Node
10004	10008	Tail of First Node
10008	20	Head of Second Node
10012	0	Tail of Second Node
10016		
...	...	

Heap example 4 — Unboxed Leaf

```
1 type tree = Branch of int * tree * tree | Leaf
2 let t = Branch (5, Branch(4,Leaf,Leaf),Leaf)
```

Stack

Addr	*Value*	
8000	10000	t
7996		
7992		
7988		
7984		
...	...	

Heap

Addr	*Value*	
10000	5	
10004	10012	Left
10008	Leaf	Right
10012	4	
10016	Leaf	Left
10020	Leaf	Right
...	...	

Heap example 4 — Boxed Leaf

```
1 type tree = Branch of int * tree * tree | Leaf
2 let t = Branch (5, Branch(4,Leaf,Leaf),Leaf)
```

Stack

Addr	*Value*	
8000	10000	t
7996		
7992		
7988		
7984		
...	...	

Heap

Addr	*Value*	
10000	5	
10004	10016	Left
10008	10012	Right
10012	Leaf	
10016	4	
10020	10012	Left
10024	10012	Right
...	...	

Heap example 5 — Cycles

```
1 type tree = Branch of int * tree * tree | Leaf
2 let rec t = Branch (5, Branch(4,Leaf,t),Leaf)
```

Stack

Addr	*Value*	
8000	10000	t
7996		
7992		
7988		
7984		
...	...	

Heap

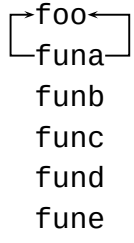
Addr	*Value*	
10000	5	
10004	10016	Left
10008	10012	Right
10012	Leaf	
10016	4	
10020	10012	Left
10024	10000	Right
...	...	

Stack Activity

- Consider the code below. The indentation is supposed to show the scope. What does the stack look like if `foo` calls `funa` calls `funb` calls `func` calls `fund` calls `fune`? Just show the static and dynamic links. The links from `funa` to `foo` have been done for you.

```
1 let foo a = ...
2    let funa x = ...
3       let funb y = ...
4       let func q = ...
5    let fund z =
6       let fune w = ...
```

Static Frame Dynamic

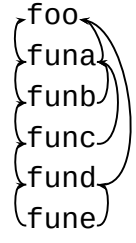


Stack Activity Answer

- Consider the code below. The indentation is supposed to show the scope. What does the stack look like if `foo` calls `funa` calls `funb` calls `func` calls `fund` calls `fune`? Just show the static and dynamic links. The links from `funa` to `foo` have been done for you.

```
1 let foo a = ...
2    let funa x = ...
3       let funb y = ...
4       let func q = ...
5    let fund z =
6       let fune w = ...
```

Dynamic Frame Static



Show the contents of the stack and the heap after the following code is evaluated. Assume that `Leaf` is unboxed.

```
1 type tree = Branch of int * tree * tree | Leaf
2 let t = Branch (5, Leaf, Leaf)
3 let rec u = Branch (7, t, u)
```



```
1 type tree = Branch of int * tree * tree | Leaf
2 let t = Branch (5, Leaf, Leaf)
3 let rec u = Branch (7, t, u)
```

Stack

Addr	*Value*	
8000	10000	t
7996	10012	u
7992		
7988		
7984		
...	...	

Heap

Addr	*Value*
10000	5
10004	Leaf
10008	Leaf
10012	7
10016	10000
10020	10012
...	...

The stack frame diagram for C languages is based on the one in *Modern Compiler Implementation in ML* by Andrew Appel, p127.